A Language for Configuring Multi-Level Specifications

Hill, G; Vickers, Steven

DOI:
10.1016/j.tcs.2005.09.065

Citation for published version (Harvard):
Hill, G & Vickers, S 2006, 'A Language for Configuring Multi-Level Specifications', *Theoretical Computer Science*, vol. 351, no. 2, pp. 146-166. https://doi.org/10.1016/j.tcs.2005.09.065

A Language for Configuring Multi-level Specifications

Gillian Hill[1] and Steven Vickers[2]

[1] Department of Computer Science, City University,
Northampton Square, London, EC1V OHB and
Department of Computing, Imperial College, London SW7 2AZ
[2] School of Computer Science, The University of Birmingham,
Birmingham, B15 2TT

Abstract. This paper shows how systems can be built from their component parts with specified sharing. Its principle contribution is a modular language for configuring systems. A configuration is a description in the new language of how a system is constructed hierarchically from specifications of its component parts. Category theory has been used to represent the composition of specifications that share a component part by constructing colimits of diagrams. We reformulated this application of category theory to view both configured specifications and their diagrams as algebraic presentations of presheaves. The framework of presheaves leads naturally to a configuration language that expresses structuring from instances of specifications, and also incorporates a new notion of instance reduction to extract the component instances from a particular configuration. The language now expresses the hierarchical structuring of multi-level configured specifications. The syntax is simple because it is independent of any specification language; structuring a diagram to represent a configuration is simple because there is no need to calculate a colimit; and combining specifications is simple because structuring is by configuration morphisms with no need to flatten either specifications or their diagrams to calculate colimits.

1 Introduction

Large complex systems are put together, or configured, from smaller parts, some of which have already been put together from even smaller parts. This paper presents a modular language that expresses the hierarchical structuring of a system from specifications of the component parts. We review briefly the mathematical framework for configuration in order to focus on the constructs of the language. Systems configuration involves specifying each of the components of the system as well as the relationship of sharing between these components. The structure of the system is therefore expressed directly and mathematically by the syntax of the configuration language, while the history of system construction is kept at a second level of mathematical structure by the accumulation of many levels of configured specifications as configuration proceeds. We propose a new and simple concept of 'instance' of a specification to manage the complexity of large systems which may require many instances of their component parts.

1.1 The Development of the Work

The motivation for our work has been to contribute to research into the modularization of systems. Our aim has been to design a language for configuring systems that is easy to use and involves concepts that should seem natural to software engineers. The language is simple because no assumptions are made about the underlying logic for specification. In earlier work we used the term 'module' to mean a 'uniquely named instance of a specification'. We now use the term 'instance', in order to avoid confusion with the use of 'module' to mean a 'composite structure wrapped up to form a single unit'. This latter use of 'module' is closer to the meaning of a configured specification.

Mathematically we were influenced by Burstall and Goguen, who gave a categorical semantics for their specification language Clear, in [2, 3]. Categorical *colimits* were used for building complex specifications in [3, 12]. We followed Oriat [9] in using colimits to express configuration in a way that was independent of any particular specification language. Oriat compared two approaches, one using diagrams and the other using a calculus of pushouts. Both in effect described the finite cocompletion of a category $\mathcal{C}$ of primitive (unconfigured) specifications.

In [13] we used instead *finitely presented presheaves*. This is a mathematically equivalent way of making a cocompletion, but leads to a different notation that very naturally describes how a configuration specifies *instances* of the component specifications, brought together with specified sharing of subcomponents. In flavour it is not unlike object-oriented languages, with the relationship between instances and specifications being analogous to that between objects and classes [8, 1] (though [13] points out some respects in which the analogy cannot be pushed too far).

As a simple example of our notation we describe, in this paper, a shop in which there are two counters sharing a single queue in which customers wait for whichever counter becomes available. We also discuss how the abstract presheaf structure is a means for describing what 'subcomponents' are, with a categorical morphism from one specification, S, to another, T, representing a means by which each instance of T may be found to bring with it an instance of S — for example, how each shop counter has a queue associated with it.

However, the approach of [13] was entirely 'flat', in that each configuration was described in terms of its primitive components. A more modular style of configuration, developed in [6], allows multi-level configuration of either primitive or previously configured components. The structure of the categorical framework is simply a hierarchy of categories, in which each configuration belongs to a level and is represented by a structured categorical diagram. Morphisms, as simple implementations between configured specifications, are allowed to cross the levels of the hierarchy. There is a notion of assignment between the instances of specifications, and in addition proof obligations are discharged. A case study, of configuration up to four levels, illustrates the expressiveness of the language. The category theory becomes somewhat deeper, with the interesting possibility of incorporating recursively defined configurations, and is still to be worked out

in detail. However, the configuration language is subject to only two simple modifications, and it is the aim of this paper to describe them.

1.2 The Structure of the Paper

In Sect. 2 the key idea of 'composites as presheaves' is introduced as an alternative to the established work on 'composites as colimits'. Presheaves provide a firm mathematical basis for the configuration language: presheaf presentations correspond to the components of a configuration and the relationship of sharing a common component; presheaf homomorphisms correspond to morphisms between configurations. In Sect. 3 we review the configuration language of [13]. Mathematically, it is formally equivalent to presenting presheaves by generators and relations, and that provides a well defined abstract semantics. Specificationally, however, one should read each configuration as specifying components and sharing. In Sect. 4 it is extended to a modular language for multi-level configuration, with two new language constructions ('basic up' morphisms, and 'indirect' morphisms). We present the case study briefly in Sect. 5, and in Sect. 6 we draw conclusions.

2 Composite Specifications as Presheaves

We gave the theoretical framework chosen for configuration in "Presheaves as Configured Specifications", [13]. Most of the technical details of the paper are due to Steven Vickers. Configuration builds composite specifications as presheaves because they express colimits in category theory. Previous research has viewed composite specifications as colimits; the approaches have varied, however, in the choice of a category with appropriate colimits. For example, the pioneering work by Burstall and Goguen on expressing the structuring of specifications by constructing the colimits of diagrams, in [2, 3], was continued in the algebraic approach to specification [5, 4, 10] and also in proof-theoretic approaches [7, 11]. All these research methods depended on the different specification logics that were used, because they constructed colimits over some cocomplete category of specifications.

A contrasting aim of configuration is to separate the specification logic of the primitive (unconfigured) specifications from their configuration. Colimits are expressed in a category of configurations which is a free cocompletion of the category of primitive specifications. There are no assumptions about the underlying logic. This more general approach allows the category of primitive specifications to be incomplete.

We followed Oriat [9] in working more generally. She models the composition of specifications by working within an equiv-category of diagrams, which is finitely cocomplete. Her equiv-category of base specifications need not be complete, however. Oriat's constructions on diagrams are shown in [13] to be mathematically equivalent to the construction of presheaves in configuration.

2.1 Presheaves

The mathematical theory of presheaves provides an alternative construction to Oriat's cocomplete category of diagrams for modelling the composition of diagrams. Formally, the category $\mathbf{Set}^{\mathcal{C}^{\mathtt{op}}}$ is the category of presheaves over a small category, $\mathcal{C}$. It follows that a *presheaf*, as an object in the category, is a functor from $\mathcal{C}^{\mathtt{op}}$ to **Set**, and a *presheaf morphism* is a natural transformation from one presheaf to another. The category $\mathbf{Set}^{\mathcal{C}^{\mathtt{op}}}$ is a free cocompletion of $\mathcal{C}$. The theory is difficult, and it is understandable that its suitability for the practical application of building specifications might be questioned. There are, however, three main reasons why presheaves express configurations precisely: when presented algebraically, a presheaf expresses the structure of a configuration; a presheaf over $\mathcal{C}$ is formally a colimit of a diagram in $\mathcal{C}$; for each morphism in $\mathcal{C}$, a presheaf presentation provides a contravariant operator from which instance reduction is defined between configurations.

The fact that $\mathbf{Set}^{\mathcal{C}^{\mathtt{op}}}$ is cocomplete means it has all small colimits. Intuitively, the fact that it is freely cocomplete means that it contains all the colimit objects and the morphisms to the colimit objects, but no more. Although expressing colimits by presheaves is more complicated theoretically than by just using diagrams, presenting presheaves algebraically simplifies the theory so that it is appropriate for configuration.

2.2 Presheaves Presented Algebraically

The key idea is that using generators and relations algebraically to present a presheaf corresponds directly to specifying components and the sharing of subcomponents in a composite system. This correspondence gives a direct physical interpretation to the configuration language.

Presheaves are presented, in detail in [13], as algebras for a many-sorted algebraic theory $\mathrm{PreSh}(\mathcal{C})$. The sorts of the theory are the objects of $\mathcal{C}$, and for each morphism $u : Y \to X$ in $\mathcal{C}$, there is a unary operator $\omega_u : X \to Y$.

The definition of an *algebra* P for $\mathrm{PreSh}(\mathcal{C})$ gives:

- for each object X of $\mathcal{C}$, a set $P(X)$, the *carrier* at X;
- for each morphism $u : Y \to X$, an operation $P(u) : P(X) \to P(Y)$ (written $x \mapsto ux$).

Algebras and homomorphisms for $\mathrm{PreSh}(\mathcal{C})$ are equivalent to presheaves and presheaf morphisms. The correspondence with configurations becomes apparent when presheaves are presented, as algebras of the algebraic theory $\mathrm{PreSh}(\mathcal{C})$, by generators and relations. We give only the main points of the correspondence:

- A *set of generators* (with respect to $\mathrm{PreSh}(\mathcal{C})$) is a set G equipped with a function $D : G \to \mathrm{ob}\ \mathcal{C}$, assigning a sort to each generator in G. In configuration the generators stand for instances of specifications. Instead of denoting the sort of a generator by $D(g) = X$, writing $g : X$ is more suggestive of declaring an instance of the specification X.

- If G is a set of generators, then a *relation* over G is a pair (e_1, e_2) (written as an equation $e_1 = e_2$) where e_1 and e_2 are two expressions of the same sort, X, say. In configuration, the expressions will describe instances of the same specification. Expressions are built out of G by applying a unary operation that corresponds to a morphism. Relations can be reduced to the form $ug_1 = ug_2$.
- A *presentation* is a pair (G, R) where G is a set of generators and R is a set of relations over G. The presheaf that is presented by (G, R) is denoted $\mathrm{PreSh}\langle G \mid R\rangle$. Presheaf presentations correspond to configurations.

Example 1. Suppose $\mathcal{C}$ is the category with two objects, X and Y, and one morphism $u : X \to Y$ (and two identity morphisms). A presheaf P over $\mathcal{C}$ is a pair of sets $P(X)$ and $P(Y)$ equipped with a function, the u operation from $P(Y)$ to $P(X)$. Suppose P is presented by generators g_1 and g_2 (both of sort Y) subject to $ug_1 = ug_2$. This is denoted by:

$$P = \mathrm{PreSh}\langle g_1, g_2 : Y \mid ug_1 = ug_2\rangle$$

Then $P(Y) = \{g_1, g_2\}$, and $P(X)$ has a single element to which u maps both g_1 and g_2. In configuration this single element is the reduction by u of g_1 and g_2.

An advantage of the correspondence with presheaves for configuration is that instead of describing an entire presheaf, by objects and morphisms, enough elements are presented to generate the rest algebraically. Although diagrams provide a simpler way of describing colimits than presheaves, the presentation by generators and relations is more natural than diagrams for expressing the configuration of components (by generators) and the sharing of components (by shared reducts).

2.3 Primitive Specifications

Configuration is over an *arbitrary* base category $\mathcal{C}$. The objects of $\mathcal{C}$ are primitive (unconfigured) specifications that, for instance, may be named after the theory presentations in the category Thpr, but are without their logical properties. For example, a theory presentation for a queue could be named as a primitive specification *Queue* in $\mathcal{C}$. The morphisms in $\mathcal{C}$ are named after the interpretations between theory presentations in mor Thpr. The category $\mathcal{C}$ is the working category for configuration: its objects are those specifications that represent the basic components of the particular system to be configured. The structure of $\mathcal{C}$ is not restricted by making it cocomplete; colimits are constructed as presheaves over $\mathcal{C}$ in a free cocompletion. This means that presheaves express configuration from primitive specifications without referring to their logical properties. Already configuration is shown to contrast with other approaches, such as [11], that work with a category of specifications over some chosen logic; presheaves are colimits whereas other approaches construct colimits of diagrams.

3 The Language for Flat Configurations

This section presents the language of [13], expressing the flat configuration of a system from primitive component parts. It assumes some fixed small category $\mathcal{C}$, whose objects stand for the primitive specifications, and constructs a category Config($\mathcal{C}$) whose objects stand for the configured specifications.

It is also important to understand the role of the morphisms. If $f : S \to T$ is a morphism (in $\mathcal{C}$ or in Config($\mathcal{C}$)), then it is intended to be interpreted as showing a way by which each instance of the specification T can be 'reduced to' an instance of S. If IT is a T instance, then we write f IT for the correspondingly reduced S instance. A typical example of what 'reduced to' means is when each instance of T — that is to say, each thing satisfying the specification T — already contains within it (as a subcomponent) an instance of S. There may be different modes of reduction. For example, if each T instance contains *two* S instances in it, then there must be two morphisms $S \to T$.

3.1 Flat Configurations

The configured specification, S, structured from instances of primitive specifications, could be expressed by:

spec S **is**
  components
    $\mathrm{IS}_1 : S_1$;
    $\vdots$
    $\mathrm{IS}_i : S_i$;
    $\vdots$
    $\mathrm{IS}_n : S_n$
  equations
    e1: $f\ \mathrm{IS}_i = g\ \mathrm{IS}_j$
    $\vdots$
endspec

The relation $e1$ states the equality between the two reducts, instances of the primitive specification T that is the common source of the morphisms f to S_i and g to S_j. The specification S_i, an object in $\mathcal{C}$, only becomes a specification in the flat world Config($\mathcal{C}$) when it is configured as *conf_*S_i and declares a formal name for a single instance of S_i:

spec *conf_*S_i **is**
  components
    $\mathrm{I}_{S_i} : S_i$
endspec

Intuitively, *conf_*S_i puts a wrapper round the named instance I_{S_i} of S_i.

Example 2. A system of counters in a post office has queues of people waiting to be served. Let *Counter* and *Queue* be specifications whose instances are actual counters and actual queuing lines. Each counter has a queue, and this instance reduction from *Counter* instances to *Queue* instances is to be represented by a morphism $i : Queue \rightarrow Counter$. The configured specification that expresses the sharing of that queue by two counters in a post office is presented as:

spec *SharingOfQueue* **is**
  components
    C_1 : *Counter* ;
    C_2 : *Counter* ;
  equations
    e1: $i\ C_1 = i\ C_2$
endspec

Although the instance of the shared queue is not declared in this general form, the expressions $i\ C_1$ and $i\ C_2$ of $e1$ each describe the instance reduct for the specification *Queue*. The specification *conf_Counter* could be configured in Config($\mathcal{C}$) by 'wrapping it up' as:

spec *conf_Counter* **is**
  components
    IC : *Counter*
endspec

3.2 Morphisms Between Flat Configurations

A morphism from one configuration, S, to another, T, is again going to represent instance reduction, showing how any instance of T can be reduced to an instance of S. We shall view this as *implementation*. Any T instance must contain all the components of S, with the correct sharing, and so provide an implementation of the specification S. The implementation is expressed by interpreting the individual components of S in T according to the assignments I $\mapsto f$ J, for I, a component of S, and J, a component of T. In addition a proof must also be given that the assignments respect the equations in S. The syntax for a configuration morphism as an implementation must therefore include both assignment of components and proof that equations hold. That proof, that is fundamental to the formal building of a system from its components, is made in the syntax of the configuration language using equations in T in a forwards or backwards direction.

Example 3. (from Ex. 2) We define two morphisms, f and g, from the configuration *conf_Counter* to *SharingOfQueue*, and a morphism, h, from *SharingOfQueue* to *conf_Counter*. f and g pick out the two counters C_1 and C_2 of *SharingOfQueue*, thus showing two ways by which a *SharingOfQueue* instance can be reduced to a *conf_Counter* instance. h describes a degenerate way in which single *conf_Counter* instance can be used to provide a *SharingOfQueue* instance, with the single counter doing all the work for two counters.

implementation f: *conf_Counter*
$\rightarrow$ *SharingOfQueue*
IC $\mapsto$ $id_{Counter}$ C_1;
endimp

implementation g: *conf_Counter*
$\rightarrow$ *SharingOfQueue*
IC $\mapsto$ $id_{Counter}$ C_2;
endimp

implementation h: *SharingOfQueue*
$\rightarrow$ *conf_Counter*
$C_1 \mapsto id_{Counter}$ IC;
$C_2 \mapsto id_{Counter}$ IC;
To check e1 of SharingOfQueue:
i $C_1 \mapsto i$; $id_{Counter}$ IC
$\hookleftarrow i$ C_2
endimp

The composition of morphisms is expressed by the notation ; . The proof that the equation $e1 : i\ C_1 = i\ C_2$ in *SharingOfQueue* is respected by the assignment of instances to *conf_Counter* is simple. The symbol $\mapsto$ denotes the assignment from the instance on the left hand side of $e1$ of *SharingOfQueue* to the instance of *conf_Counter*. Finally the symbol $\hookleftarrow$ denotes the assignment from i C_2 on the right hand side of $e1$ in *SharingOfQueue* to i ; $id_{Counter}$ IC in *conf_Counter*.

The morphism h makes the point that the mathematics of colimits as used for specification can specify equalities but not inequalities.

4 The Language for Multi-level Configurations

The aim of this section is to extend the configuration language by modularity to express the hierarchical structuring of multi-level configurations, independently of any logic. The syntax of the modular configuration language directly expresses the structure of a system, so that the user of the configuration language is able to record the history of configuration in easily understood amounts.

Configuration offers a semantics for the structuring of specifications which is new in two respects. The first is that flattening can be avoided because configurations are isomorphic to their flattened form. The second respect is that the manipulations do not rely on a flattened form even existing. The language allows morphisms to be defined with 'relative' flattening down a few levels in the hierarchical configuration but without necessarily reaching a primitive level. To match this, [6] does not construct the mathematical workspace inductively, starting

with the primitive level and working up, but instead offers an axiomatic approach that identifies the structure needed to interpret the language constructs. Potentially then, the workspace can contain configurations of infinite depth and give meaning to recursively defined configurations.

4.1 The Objects and Morphisms in the Configuration Workspace

Providing a new mathematical semantics for structuring multi-level specifications in a categorical workspace leads to a new engineering style of manipulation for the specifications. The primitive and configured specifications are collected together in a single category and configuration becomes a construction that can be applied with arbitrary objects and morphisms. Since S and *conf_S* are now objects in the same category they are assumed to be isomorphic, and this isomorphism leads to the extra syntactic features of *basic up* and *indirect* morphisms in the multi-level language.
Objects are either primitive or configured.
Primitive objects are drawn from a category $\mathcal{C}$.
Configured objects use the keywords **spec** and **endspec** as before to put together components with sharing. However, now their component specifications may themselves be either primitive or configured, possibly with some of each.
Morphisms may be defined between any objects in the workspace, and are needed to construct new objects or to prove that objects are equivalent. Again, they represent a contravariant notion of instance reduction, that gets an instance of the source specification from an instance of the target.
Primitive morphisms from $\mathcal{C}$ are between primitive specifications.
Configuration morphisms are defined as in Sect. 3.

However, new morphisms are needed to make any configuration S isomorphic to the configured specification *conf_S* that declares an instance of S.

4.2 Basic up morphisms

These morphisms arise from the need for a morphism from $S \rightarrow$ *conf_S*. Suppose I_S is declared as the component in *conf_S*. Our syntactic device is to use that instance name also as the name of the morphism, $I_S : S \rightarrow$ *conf_S*. If IS: S is a component in a configuration T, then as in Sect. 3, we can define a configured morphism

implementation h: *conf_S* $\rightarrow T$
$\mathrm{I}_S \mapsto id_S$ IS
endimp

The morphism h can be composed with the isomorphism $S \rightarrow$ *conf_S* to get a morphism f from S to T. Again we apply the device of using the instance name IS as the name of this composite morphism, $IS : S \rightarrow T$, and this is the most general form of what we shall call a *basic up* morphism. Note that S may be either primitive or configured.

4.3 Indirect morphisms

These arise from the morphism *conf_S* $\rightarrow S$ and are defined as *indirect implementations* that use the keyword **given**. This syntax provides a formal name for an instance in the target specification of the morphism:

implementation f: $T \rightarrow S$
given instance IS: S
⋮
endimp

Here the middle, omitted, part is just the usual format (as before) for the body of a configuration morphism. The instance name provided can be taken as defining an anonymous configuration which is isomorphic to *conf_*S:

spec - - - **is**
  components
    IS : S
endspec

The indirect definition of f supplies the data for a morphism from T to this anonymous configuration. This is then composed with the isomorphism *conf_*S $\rightarrow S$ to give the indirect morphism $f : T \rightarrow S$. Again indirect morphisms arise from the need to have every S isomorphic to *conf_*S. The isomorphism *conf_*S $\rightarrow S$ can itself be denoted using the 'given' notation.

4.4 Morphisms between multi-level configurations

We have defined morphisms from configured specifications to primitives. We also need to define them between configured specifications.

Example 4. (from Ex. 2) Second level and first level configurations illustrate two ways of making a post office with three counters and one shared queue:

spec *ExtendedShop* **is**
  components
    $\mathrm{C_1QC_2}$: *SharingOfQueue* ;
    $\mathrm{C_3}$: *Counter* ;
  equations
    e1: i $\mathrm{C_3} = i$ **;** C_1 $\mathrm{C_1QC_2}$
endspec

The morphism C_1 is a basic up morphism.

spec *NewShop* **is**
  components
    C_1 : *Counter* ;
    C_2 : *Counter* ;
    C_3 : *Counter* ;
  equations
    e1: $i\ \mathrm{C}_1 = i\ \mathrm{C}_2$;
    e2: $i\ \mathrm{C}_1 = i\ \mathrm{C}_3$
endspec

These configurations are isomorphic, but the isomorphism g: *ExtendedShop* $\rightarrow$ *NewShop* cannot be defined except indirectly, with **given**. The syntax of the indirect implementation, g, also uses a keyword **where** to introduce a locally defined morphism, f: *SharingOfQueue* $\rightarrow$ *NewShop*.

implementation g: *ExtendedShop*
  $\rightarrow$ *NewShop*
given instance INS: *NewShop*
$\mathrm{C}_1\mathrm{QC}_2 \mapsto f$ INS ;
$\mathrm{C}_3 \mapsto C_3$ INS ;
where
  implementation f: *SharingOfQueue*
    $\rightarrow$ *NewShop*
  $\mathrm{C}_1 \mapsto \mathrm{C}_1$;
  $\mathrm{C}_2 \mapsto \mathrm{C}_2$;
  *To check e*1 *of SharingOfQueue*:
  $i\ \mathrm{C}_1 \mapsto i\ \mathrm{C}_1$
    $= i\ \mathrm{C}_2$ *by e*1 *of NewShop*
    $\hookleftarrow i\ \mathrm{C}_2$
  endimp
*To check e*1 *of ExtendedShop*:
$i\ \mathrm{C}_3 \mapsto i$ **;** C_3 INS
  $= i$ **;** C_1 INS *by e*2 *of NewShop*
  $= i$ **;** C_1 **;** f INS
  $\hookleftarrow i$ **;** $C_1\ \mathrm{C}_1\mathrm{QC}_2$
endimp

The proof for equation $e1$ of *ExtendedShop* uses the fact that C_1 INS $= C_1$ **;** f INS. This comes directly out of the definition of f, from $\mathrm{C}_1 \mapsto \mathrm{C}_1$.

5 A Case Study

We use the new configuration language in a case study, based on an example of Oriat's [9], to express alternative configurations for the theory of rings. In [6] the aim of the case study is to compare Oriat's method of composing specifications,

by constructing the pushouts of diagrams, with the method of configuration. Since in configuration both specifications and their diagrams express algebraic presentations of presheaves, and finitely presented presheaves express colimits, the need to construct pushout diagrams is bypassed. Since equivalence between configurations can be proved textually, Oriat's need to flatten diagrams (to construct their colimits) and to complete diagrams before normalizing them can also be bypassed.

5.1 Building Flat Configurations from Primitive Specifications

The theory presentations and theory morphisms that underly the primitive specifications for the components used to configure a ring are expressed in the style of Z schemas. As in Sect. 2.3 we use the name of each theory presentation, forgetting its logical properties, to identify a primitive specification. The simplest component of the mathematical structure of a ring expresses a single sort s.

Asort[s]

The schema *Bin-op* specifies a sort, also called s, and a binary operator *op*:

Bin-op[s]

$op : s \times s \to s$

The theory morphism $s : Asort \to Bin\text{-}op$ maps the sort of *Asort* to the sort of *Bin-op*. The schema for the structure of a monoid is:

Monoid[s]

$\times : s \times s \to s$

$1 :\to s$

$\forall x, y, z : s \;.\; (x \times y) \times z = x \times (y \times z)$

$\forall x : s \;.\; 1 \times x = x$

$\forall x : s \;.\; x \times 1 = x$

The theory morphism $b : Bin\text{-}op \to Monoid$ maps the sort of *Bin-op* to the sort of the monoid, and the operator *op* of *Bin-op* to the operator $\times$ in the monoid. The theory presentation for an Abelian group is formed from *Monoid* by adding an inverse function and the property of commutativity for the binary operator, $+$. The theory morphism m maps the operator $\times$ of *Monoid* to the operator $+$ of *Abel-group* and the constant 1 of *Monoid* to the constant 0 of *Abel-group*.

Abel-group[s]

$+ : s, s \to s$
$0 :\to s$
$inv : s \to s$

$\forall\, x, y, z : s \;.\; (x + y) + z = x + (y + z)$
$\forall\, x : s \;.\; 0 + x = x$
$\forall\, x : s \;.\; inv(x) + x = 0$
$\forall\, x, y : s \;.\; x + y = y + x$

Finally the schema *Distributive* specifies two binary operators that are related by the property of distributivity. There are two morphisms from *Bin-op* to *Distributive*: the morphism m_+ maps *op* to $+$; the morphism $m_\times$ maps *op* to $\times$. The axioms for the distributive structure express both left and right distributivity for $\times$ over $+$.

Distributive[s]

$+ : s, s \to s$
$\times : s, s \to s$

$\forall\, x, y, z : s \;.\; x \times (y + z) = (x \times y) + (x \times z)$
$\forall\, x, y, z : s \;.\; (y + z) \times x = (y \times x) + (z \times x)$

In the text of the configured specifications we use abbreviations for the instance names. Of four equivalent specifications for the flat configuration of a ring the following is the most compact:

spec Ring1 **is**
  components
    M : Monoid ;
    A : Abel-group ;
    D : Distributive ;
  equations
    e1: b **;** m A = m_+ D ;
    e2: b M = $m_\times$ D
endspec

The specification *Ring1* describes the sharing of the boolean operators explicitly. The instance reduct b **;** m A gives the binary operator for addition, derived by reduction from the instance A of *Abel-group*. The instance reduct b M is the operator for multiplication, derived by reduction from the instance M of *Monoid*. That is, $e1$ describes the sharing of the addition instance of *Bin-op*, and $e2$ describes the sharing of the multiplication instance of *Bin-op*.

5.2 Natural Uses of Modularization

In Oriat's language of terms, all colimits of representative diagrams are pushouts. In the configuration language, modularization is only used if required specificationally: it is not imposed by pushout terms. Configurations that correspond to

Oriat's modular constructions of a ring are built in [6]. Two of these are more natural because, although they are built by adding distributivity to a pseudo-ring, neither requires the construction of an extra configuration for the pair of binary operators. Together with the flat *Ring1*, we select these modularized configurations as the ideal configurations for a ring. *Ring4*, a fourth-level specification, illustrates the flexibility of our language by expressing the sharing of each instance of the binary operator in an equation. The history of the configuration is presented first in the three lower-level configurations.

spec Pair_Bin-op_and_Asort **is**
  components
    a : Bin-op ;
    m : Bin-op ;
  equations
    e1: s a = s m *sharing the instance* s
endspec

spec Pair_Bin-op_Asort_and_Monoid **is**
  components
    M : Monoid ;
    ams : Pair_Bin-op_and_Asort;
  equations
  e1: b M = m ams
endspec

spec Pair_Bin-op_Asort_Monoid_and_Abel-group **is**
  components
    amsM : Pair_Bin-op_Asort_and_Monoid;
    A : Abel-group ;
  equations
    e1: a **;** *ams* amsM = b **;** m A
endspec

spec Ring4 **is**
  components
    D : Distributive ;
    amsMA : Pair_Bin-op_Asort_Monoid_and_Abel-group ;
  equations
    e1: $m_{\times}$ D = m **;** *ams* **;** *amsM* amsMA ; *sharing the instance* m
    e2: m_{+} D = a **;** *ams* **;** *amsM* amsMA *sharing the instance* a
endspec

6 Conclusions

We thank the reviewers for inspiring us to improve the paper. Our goal has been to introduce, independently of specification language, a modular configuration language that expresses the construction of large complex systems from their component parts, with specified sharing. We have already presented in [13] a configuration language based on components and sharing that is independent of specification language. It has an abstract semantics using presheaves that is mathematically equivalent to the diagrammatic approach of [9]. However, it is limited to flat configurations: it has no modularity and is unable to express any further structuring to multi-level configurations. The modularity here, avoiding the need to flatten structured specifications, has been achieved categorically in [6] by having explicit isomorphisms between unflattened configurations that would become equivalent when flattened. Linguistically it works by the use of two new constructions, the basic ups and the indirect configuration morphisms, whose interpretation provides those isomorphisms. Although the configuration language has been presented with a detailed case study in [6], more work is required on the semantics of the language. The need to avoid the absolute flattening of configured specifications to a primitive level suggests that a hierarchical workspace of infinite depth should be constructed with the potential to deal with recursively defined configurations.

References

1. Booch, G.: Object-Oriented Analysis and Design. The Benjamin Cummings Publishing Company, Inc. second edition (1994)
2. Burstall, R. M., Goguen, J. A.: Putting Theories Together to Make Specifications. Proc.of the 5th. International Joint Conference on Artificial Intelligence, Cambridge, Mass. (1977) 1045–1058
3. Burstall, R. M., Goguen, J. A.: The Semantics of Clear, A Specification Language. Abstract Software Specifications, LNCS 86 (1979)
4. Ehrig, H., Fey, W., Hansen, H., Lowe, M., Papisi-Presicce, F.: Algebraic Theory of Modular Specification Development. Technical report Technical University of Berlin (1987)
5. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Springer-Verlag (1985)
6. Hill, G.: A Language for Configuring Systems. PhD Thesis, Department of Computing, Imperial College, University of London (2002)
7. Maibaum, T. S. E., Sadler, M. R., Veloso, P. A. S.: Logical Specification and Implementation. Foundations of Software Technology and Theoretical Computer Science, LNCS **181** Springer-Verlag (1984)
8. Meyer, B.: Reusability. IEEE Software, (1987) 50–63.
9. Oriat, C.: Detecting Equivalence of Modular Specifications with Categorical Diagrams. Theoretical Computer Science, 247 (2000) 141–190
10. Sannella, D., Tarlecki, A.: Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. Acta Informatica 25 (1988) **3** 233–281
11. Srinivas, Y. V., Jellig, R.: Formal Support for Composing Software. Proc. of Conference on the Mathematics of Program Construction, Kloster Irsee, Germany July (1995) Kestrel Institute Technical Report KES.U.94.5
12. Veloso, P. A. S., Fiadeiro, J., Veloso, S. R. M.: On local modularity and interpolation in entailment systems. Inform. Proc. Lett. 82(4)(2002) 203 – 211
13. Vickers, S., Hill, G.: Presheaves as Configured Specifications. Formal Aspects of Computing 13 (2001) 32–49